SPV User Manual and Quick Start Guide

Kaile Su, Yinyin Xiao, Weiya Yue and Qingliang Chen
Department of Computer Science, Sun Yat-sen University Guangzhou, P.R. China

January 18, 2007

SPV Version 3.8.2, January 2007

Welcome to SPV:

a Security Protocol Verifier

SPV is a robust software package used primarily for automatic verification of complex security protocols with an unbounded number of sessions. In detail, the function of **SPV** is the following:

GIVEN: Input a formal description of a particular protocol and some security properties, which are to be verified.

RESULT: Output whether the properties are satisfied by the protocol, the number of variables and constrains generated in the verification process, and the run time of the process.

SPV is the implementation of Logic of Local Sessions (LLS), which is a new protocol logic based on a quite natural semantic model called Instantiation Space. The advantages of **SPV** are as follows:

- **SPV** can deal with complex message formats with arbitrarily nested encryptions by public, private, shared and hash keys as well as freshly generated keys, due to the flexibility of LLS.
- **SPV** can be used to verify complex security properties such as 'Alice observes (knows) Bob observes (knows) Alice said something', and the SECRET and PRECEDES in CAPSL.
- **SPV** has been applied to automatically verify a lot of interesting and important properties for quite complex security protocols like Kerberos V5 and the SET protocol. For example, for the complex SET purchase phase protocol, **SPV** takes only several hours to generate eighty thousand rules and then verify many concerned specifications in several seconds.
- **SPV** is very easy to be used. For example, the input language of **SPV** is just the actual definitions of the protocols and is user-friendly since it has more intimacy to its original forms. Moreover, users can define macros by themselves, which simplifies the input process for big protocols.
- **SPV** is a fully automatic verifier compared to semi-automatic verifiers like theorem prover Isabelle.

The newest version of **SPV** is 3.8.2. It's now orders of magnitude faster than the first version. It has been designed and constructed robustly. **SPV** is based on SAT solvers. Particularly, we used the effective SBSAT (http://www.cs.uc.edu/~weaversa/SBSAT.html) as the underlying SAT solver. Its main execution paths are:

1. Analyze the input file, and convert it into various related data-structures.
2. Generate all the variables and properties of agents' behavior. The number of them is decided by the complexity of the protocol, which is decided by message structures, agents involved and protocol steps.
3. Turn the verification problem into an instance of the Satisfiability problem, and utilize the powerful SAT solver SBSAT to compute the specification.

The schematic depiction of controllable execution paths of **SPV** is showed in Figure 1.

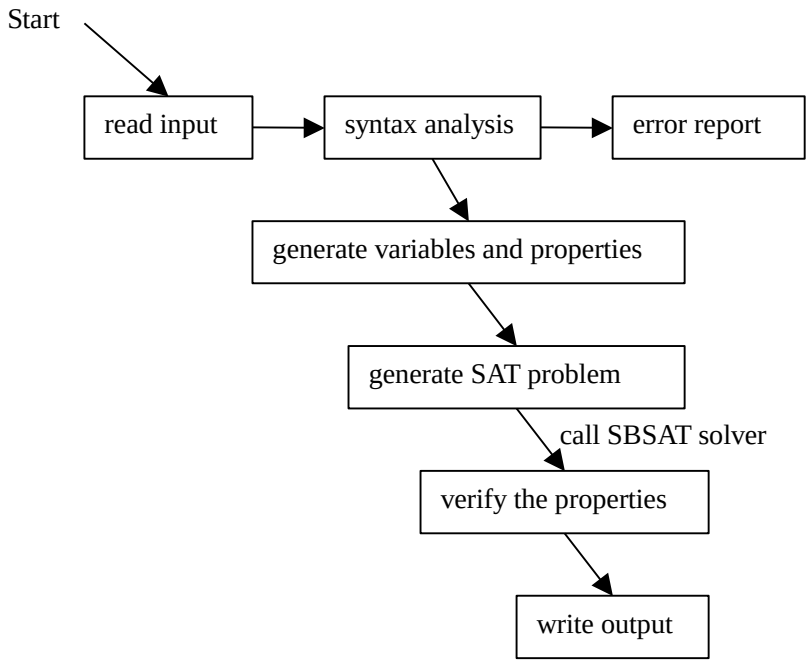

Figure 1: Schematic depiction of controllable execution paths of **SPV**

In the whole process, most of the time is spent on the generations of variables and properties, and the time for verification is just within seconds. So there will be a lot of scope for optimizations and scaling to utilize the power of SBSAT. We will also explore heuristics necessary to make **SPV** more efficient.

The **SPV** can be successfully running on Unix based platforms with compiler gcc 3.2.2 or higher versions. A lot of experiments have been conducted on a PC with AMD Opteron242, 2GB DDR memory, running on Redhat Linux 7.0 with gcc 3.2.2.

Contents

1 About the manual

The manual has two parts: Sections 2 to 4 are written to get the novice acquainted with the use of **SPV** quickly; the following sections, beginning with Section 5, provide details of pre sections and advanced features of **SPV** needed for an accomplished user.

2 Quick Start – Getting SPV ready to run

This and the following two sections are intended to provide enough information to begin using **SPV** successfully.

2.1 Hardware Requirements

Currently, **SPV** requires a Unix style operating system with a c++ compiler (gcc 3.2.2 or higher versions). All examples require at least 32MB of RAM beyond the requirements of the operating system. Disk requirements depend on the operating system but at least 50MB of free space is required. A lot of experiments have been conducted on a PC with AMD Opteron242, 2GB DDR memory, running under Redhat Linux 7.0 with gcc 3.2.2.

By default, during execution, **SPV** is allocated as much RAM as it needs, if available. The amount of memory requested by **SPV** can be limited only indirectly by changing, for example, the number of variables and properties it maintains in the cache or the size of the pools for different stacks. There is no other option to limit the amount of memory it is allocated. Experiments confirm that the amount of memory requested linearly follows the size of the problem being solved.

2.2 Getting SPV

SPV is available for download from the following website:

http://www.cs.sysu.edu.cn/~skl/spv.htm

SPV may also be obtained by email request to isskls@mail.sysu.edu.cn, xyycx1983@163.com, ywyisywd@163.com, or Tsingliangchen@hotmail.com. The distribution comes in the following form: a compressed file named **spv.zip**. We choose this format because this kind of compressed file is the most compatible one on MS Windows based platforms. Therefore, users can transfer the **SPV** file between Unix and MS Windows platform conveniently.

2.3 Installing SPV

These instructions are only for installing **SPV** on computers running Unix. Instructions for Windows machines will be supplied in a future release.

Become **root** (This step may not be necessary). This entails knowing the superuser password. At the command line prompt, issue the command **su** and enter the superuser password when requested to do so.

If the file **spv.zip** is in CDROM, USB hard disk, or in the partition of other Operation Systems,

such as MS Windows, you should mount the relative device first. Take the CDROM for example: after you insert the CDROM into the CDROM drive, you should mount the drive, usually on **/mnt/cdrom**, using the following command:

$ mount /dev/cdrom /mnt/cdrom

If this command fails, find a suitable mount point in place of **/mnt/cdrom** or find the correct **/dev** for the CDROM(for example, **/dev/scd0**) or both. If this still fails, consult a system administrator. The following assumes the above command succeeded. Change directory to the place where **SPV** is to be installed (for example **/usr/local**), and copy the contents of the CDROM to the current directory using the following commands:

$ cd /usr/local

$ cp -r /mnt/cdrom/spv.zip .

where the '.' is part of the command and means current directory. Use the **umount** command to unmount the CDROM as follows:

$ umount /mnt/cdrom

If the **spv.zip** is already in the Unix file system, move it to the directory in which **SPV** is to reside. For example, if the target directory is **/usr/local** and **spv.zip** exists in the home directory of a user named **agent** then issue the commands:

$ mv /agent/spv.zip /usr/local

When the file **spv.zip** is in the right place, change directory to the place and unzip **spv.zip** using the following commands:

$ cd /usr/local

$ unzip spv.zip

You may remove the **spv.zip**, if you wish, with:

$ rm spv.zip

The result of the above commands is that all files of the **SPV** package are in the directory **/usr/local/spv**. Now you can run the **SPV** program.

3 Quick Start – Running SPV

This section illustrates how a user can fine-tune a run of **SPV** on a given input. It is assumed that the **SPV** has been set as per Section 2.3. This makes the **SPV** accessible to everyone from every directory. All examples in this manual are part of the **SPV** distribution and may be found in the **…/spv/res** directory.

3.1 Command line

Become root as in Section 2.3 (This step may not be necessary). Change to the directory containing the **SPV** files, called the root directory of **SPV**. If you followed the instructions in Section 2.3, this is accomplished with the following command:

$ cd /usr/local/spv

Now you can run the **SPV** in this directory with:

$ ./s inputfile [>outputfile]

where the parameter **inputfile** with the extension name '.p' is the input file of **SPV** (We will introduce the input file in Section 3.2), and the optional parameter **outputfile** which can be any types of text file is the output file of **SPV**. Notice that the '>' means output redirection. If you choose the **outputfile** parameter, the running result will write in the output file, or else it will print on terminal.

We have verified a lot of complex security protocols, including those from the Security Protocols Open Repository. For more details, please refer to the Section 7. All examples of input files can be found in the **…/spv/res** directory. You can run **SPV** on them. For example, the Needham-Schroeder protocol is described in the input file **ns.p,** and you can verify it with the command:

$ ./s res/ns.p

Now the result will be printed on terminal. If you intend to save the result in an output file, such as **/res/ns_rerult.txt**, use the following command:

$ ./s res/ns.p >res/ns_result.txt

Remark: At the beginning of running, **SPV** will analyze whether the syntax of input file is legal according to the definition of input file. If there are no syntax errors, the program will accept the input file, verify the protocol, and finally output the result; otherwise it will stop and output the error message, and you can revise the input file according to the message.

3.2 Input file

An input file is a formal protocol description accepted by **SPV**. This section illustrates the structure of

the input file of **SPV**, and then an example is given. For more detailed information for the input formats, please refer to Section 5.

3.2.1 Structure

The accepted input file is defined very clearly and intuitively. The necessary parts of it are as follows:

- **#variable:** Variable definitions begin with **#variable** to define all the variables needed in the protocol, such as agent, nonce and key, etc.
- **#initialize:** Initialization section begins with **#initialize** to initialize all relations among the variables defined above.
- **#protocol:** The protocol body begins with **#protocol** to describe the protocol.
- **#role_assumption:** The role definition begins with **#role_assumption** to describe that agents intends to play the roles of the protocol in a local run.
- **#specification:** The specifications which users want to verify begin with **#specification**.

Besides, **SPV** has given more features and flexibilities in the user-interface for the advanced users, who have more understanding about the Logic of Local Sessions:

- **#macro:** The macro definitions begin with **#macro** to define macros, which simplifies the input process for big protocols.
- **#define:** The **#define** section begins with **#define** to allow users to define their assumptions a little different from the default situation in the Logic of Local Sessions.
- **#goal**：The specifications in Common Authentication Protocol Specification Language (CAPSL) format begins with **#goal**.

Remark: **SPV** allows users add **comments** for the input file to improve its readability. Just like the format in common programming languages, the comments begin with the symbol **'//'** and can be put in any positions of input file without influencing other contents.

3.2.2 Example: ns.p

In this part, an entire simple example of input file **ns.p** is given, which is designed for the famous Needham-Schroeder protocol.

```
// ns.p: Needham-Schroeder protocol
// define variables
#variable
agent : alice, bob;          // actual agents
agent_term: A, B;            // roles in protocol
nonce : na, nb;              // actual challenge numbers
nonce_term: Na, Nb;          // nonce terms in protocol
```

```
publicKey_term: ka, kb;          // public key terms in protocol
privateKey_term: _ka, _kb;       // private key terms in protocol

// initialize variables
#initialize
name(alice, A);                  // alice plays role A
name(bob, B);                    // bob plays role B
name(na, Na);                    // na is the actual nonce of Na
name(nb, Nb);                    // nb is the actual nonce of Nb
nonce(A, Na);                    // Na is generated by A
nonce(B, Nb);                    // Nb is generated by B
inverse_key(A, ka, _ka);         // ka and _ka is A's public key and private key, respectively
inverse_key(B, kb, _kb);         //kb and _kb is B's public key and private key, respectively

// protocol description
#protocol
A, B : cryp(kb, (A, Na));        // A sends the message term (A, Na) encrypted by kb to B
B, A : cryp(ka, (Na, Nb));       // B sends the message term (Na, Nb) encrypted by ka to A
A, B : cryp(kb, Nb);             // A sends the message term Nb encrypted by kb to B

// role assumption for the agents
#role_assumption
sees(alice, nb) + said(alice, nb) > role(alice, nb, A);
sees(bob, na) + said(bob, na) > role(bob, na, B);

// the general specifications
#specification
know(alice, (said(bob, na)));    // alice knows that bob has said na
know(bob, (said(alice, nb)));    // bob knows that alice has said nb
// alice knows that bob has known that alice has said nb
know(alice, know(bob, (said(alice, nb))));
// bob knows that alice has known that bob said na
know(bob, know(alice, (said(bob, na))));

// the CAPSL specifications
#goal
```

secret Na; **// SECRET goals**

secret Nb;

precedes A : B | Na, Nb; **// PRECEDES goals**

precedes B : A | Na, Nb;

3.3 Outputs

If the users run an input file as per Section 3.1, the result will be given on terminal or in an output file. This section gives a brief introduction for the output of **SPV**, and then an example is given. For more detailed information for the output formats, please refer to Section 5.

3.3.1 Error message and normal outputs

- If the input file exists some syntax errors, such as that the variables lack definitions, or the format is wrong, the output will print the error messages. For example, for the former error, the output will print:

  is not defined

  Then users can revise the input file according to the hints.
- If there are no syntax errors in the input file, we can get a normal output. It is composed of the required running time, the number of variables and constrains, and the verification results of each specification.

3.3.2 Example: results of ns.p

We have conducted the input file **ns.p** in Section 3.2.3 on a PC with AMD Opteron242, 2GB DDR memory, running under Redhat Linux 7.0 with gcc 3.2.2. This section shows the results.

// result of ns.p

We spend 42.1 seconds to generate all the constrains!

the number of variables is 3548 and the number of constrains is 6296

==========********==========**

know(alice, said(bob,na))

OK

=======

We spend 0.04 seconds to verify this specifications!

know(bob, said(alice,nb))

OK

=======

We spend 0.04 seconds to verify this specifications!

know(alice, know(bob, (said(alice, nb))))

OK

=======

We spend 0.38 seconds to verify this specifications!

know(bob, know(alice, said(bob,na)))

NO

=======

We spend 0.42 seconds to verify this specifications!

==========*******secrecy*************==========**

----------secrecy(alice, bob, local(alice, na, Na))----------

OK

We spend 0.34 seconds to verify this specifications!

==========*******secrecy*************==========**

----------secrecy(bob, alice, local(bob, nb, Nb))----------

FAIL

We spend 0.36 seconds to verify this specifications!

==========*******precedes*************==========**

---reach(bob, nb, B, 3) * equal(alice, local(bob, nb, A)) * equal(na, local(bob, nb, Na))) > (role(alice, na, A) * equal(bob, local(alice, na, B)) * equal(nb, local(alice, na, Nb)) * equal(local(bob, nb, Nb), local(alice, na, Nb)))---

FAIL

We spend 0.05 seconds to verify this specifications!

==========*******precedes*************==========**

---reach(alice, na, A, 3) * equal(bob, local(alice, na, B)) * equal(na, local(alice, na, Na))) > (role(bob, na, B) * equal(alice, local(bob, na, A)) * equal(na, local(bob, na, Na)) * equal(local(bob, na, Nb), local(alice, na, Nb)))----

OK

We spend 0.05 seconds to verify this specifications!

As can be seen, the specification **know(bob,know(alice,said(bob,na)))** does not hold, which shows the cause for the famous Lowe attack.

4 Quick Start – Getting more help quickly

- Check out the **SPV** Web Page: http://www.cs.sysu.edu.cn/~skl/spv.htm
- Email us:

  Kaile Su isskls@mail.sysu.edu.cn

  Yinyin Xiao xyycx1983@163.com

  Weiya Yue ywyisywd@163.com

  Qingliang Chen Tsingliangchen@hotmail.com

5 Reference – Input formats

This section illustrates the format of the input file of **SPV**, and how to write an appropriate input file for a given protocol.

5.1 Basic definitions

The basic definitions **#variable**, **#initialize**, **#protocol**, **#role_assumption** are necessary for every legal input file.

5.1.1 #variable

The main variable types in **#variable** section are as follows:

- **agent:** The principals which play the roles of protocol.
- **agent_term:** The roles in protocol.
- **nonce:** The concrete challenge messages generated by agents.
- **nonce_term:** The nonce terms described in protocol.
- **guessable_lable_message:** The time stamps generated by agents.
- **guessable_lable_message_term:** The time stamp terms described in protocol.
- **publicKey_term:** The public key terms of roles.
- **privateKey_term:** The private key terms of roles.
- **shareKey_term:** The share key terms of roles.
- **generalMessage_term:** The general plain text terms in protocol.
- **hash:** The hash key of roles.
- **xor:** The xor key of roles. In **SPV**, xor operation is viewed as a kind of encryption, so **xor** is defined as a special key.

Remark: For convenience, **SPV** does not separate keys of agents and the key terms of protocol, as well as the general message and the general message term. This simplification does not influence the verification. Besides, **hash** and **xor** are special keys, which can be used directly. Therefore, users needn't define them in the **#variable** section.

The format of variable definition is:

#variable

variable type$_1$: variable name$_{11}$, variable name$_{12}$, …;

variable type$_2$: variable name$_{21}$, variable name$_{22}$,…;

…

where the **variable name$_{ij}$**(i =1,2, …, m, j = 1,2, …, n) is the jth variable declaration of the ith variable type. All the variable names are strings.

for example:

#variable

agent: alice, bob;

agent_term: A, B;

5.1.2 #initialize

The main operations with the accepted types of parameters in **#initialize** section are as follows:

- **name(agent name, agent_term name):** Bind the agent to the agent term, indicating that the agent plays the role of agent term.
- **name(nonce name, nonce_term name):** Bind the nonce to the nonce term, indicating that the nonce term is instantiated by the concrete nonce.
- **name(guessable_lable_message name, guessable_lable_message_term name):** Bind the guessable label message to the guessable label message term, indicating that the latter is instantiated by the former one.
- **nonce(agent_term name, nonce_term name):** Indicate that the nonce term is generated by the agent term in protocol.
- **guessable_lable_message(agent_term name, guessable_lable_message_term name):** Indicate that the guessable label message term is generated by the agent term in protocol.
- **initial_general_message(generalMessage_term name, agent_term name):** Indicate that at the beginning, the general message term is sent by the agent term in protocol.
- **inverse_key(agent_term name, publicKey_term name, privateKey_term name):** Bind the public key term, private key term to agent term, showing that both of the key terms belongs to the agent term, and the private key term corresponds with the public key term.
- **share_key(agent_term name, agent_term name$_1$, shareKey_term name$_2$):** Bind the share

key term to the agent terms, showing that the key term is shared between the agent terms.

The format of initialization in **#initialize** section is:

#initialize

initialization operation$_1$;

initialization operation$_2$;

…

for example:

#initialize

name(alice, A);

name(bob, B);

5.1.3 #protocol

cryp is an important operation in **#protocol** section. We define the following auxiliary variable types first:

- **key ::= publicKey_term | privateKey_term | shareKey_term | hash**
- **M ::= agent_term | nonce_term | guessable_lable_message_term | generalMessage_term | publicKey_term | privateKey_term | shareKey_term | M, M | cryp(key, (M)) | cryp(xor, (M), (M))**

Intuitively, **M** corresponds with all the message terms. And the definition of **cryp** included in the recursive definition of **M** is as follows:

- **cryp(key name, (M name)):** Encrypt message term **M name** with **key name**.
- **cryp(xor, (M name$_1$), (M name$_2$)):** Apply the xor operation on the two message terms.

Remark: If the message terms in **cryp** are not concatenation message terms and there are no confusions, then the parentheses of them can be omitted.

The format of protocol body in **#protocol** section is:

#protocol

sender$_1$, receiver$_1$: M name$_1$;

sender$_2$, receiver$_2$: M name$_2$;

…

where **sender$_i$** and **receiver$_i$** are agent terms, **M name$_i$** is message term(i = 1,2, …, n), and **sender$_i$** sends **M name$_i$** to **receiver$_i$** in the protocol.

For example:

#protocol

A, B: cryp(kb, (A, Na));

B, A: cryp(ka, (B, Nb));

5.1.4 #role_assumption

Role assumption is an important axiom in Logic of Local Sessions. In the logic, we assume every nonce or guessable label message is used in at most one agent's local run in a protocol. Therefore, nonce and guessable label message can be viewed as the label message, which could denote an agent's local run. The role assumption shows that if an agent can see (get) or has said (sent) a label message, then the agent plays a role of the protocol in a local run denoted by the label message.

The following auxiliary variable types are used in the role assumption:

- **label ::= nonce | guessable_lable_message**
- **m ::= agent | nonce | guessable_lable_message | m, m | local(agent, label, M)**

Intuitively, **m** corresponds with all the actual messages. The definition of **local** in **m** is as follows:

- **local(agent name, label name, (M name)):** It returns an actual message which corresponds the message term **M name** in **agent name**'s local run denoted by the **label name**.

The main atomic formulas in the role assumption are:

- **sees(agent name, (m name)):** Indicate that the agent can get the actual message **m name**.
- **said(agent name, (m name)):** Indicate that the agent has sent the actual message **m name**.
- **role(agent name, label name, agent_term name):** Indicate that the agent plays the role(the 3rd parameter) in the local run denoted by the label message(the 2nd parameter) in protocol.

Remark: The parentheses of the message term **M name** or the message **m name** above can be omitted as the situation in **cryp.** We will not remind this simplification in the remaining contents.

The format of **#role_assumption** section is as follows:

#role_assumption

sees(agent name$_1$, label name$_1$) + said(agent name$_1$, label name$_1$) > role(agent name$_1$, label name$_1$, agent_term name$_1$);

sees(agent name$_2$, label name$_2$) + said(agent name$_2$, label name$_2$) > role(agent name$_2$, label name$_2$, agent_term name$_2$);

…

where the symbol **'+'** means 'or' and **'>'** means 'implication'. Besides, **SPV** uses the symbol '*' to denote 'and'.

For example:

#role_assumption

sees(alice, nb) + said(alice, nb) > role(alice, nb, A);

5.1.5 #specification

The main atomic formulas used in **#specification** section are **sees**, **said**, **role**, **reach** and **equal**:

- **atomic_formula ::= sees(agent, (m)) | said(agent, (m)) | role(agent, label, agent_term) | reach(agent, label, agent_term, l) | equal((m), (m))**

where **l** is nature number, and the definition of **reach** and **equal** is:

- **reach(agent name, label name, agent_term name, l name):** Indicate that the agent plays the role (the 3rd parameter) and reaches step (**l name** -1) of local run denoted by the label message(the 2nd parameter) in protocol. In particular, **reach(agent name, label name, agent_term name, 0)** is equivalent to **role(agent name, label name, agent_term name).**
- **equal((m name$_1$), (m name$_2$)):** Indicate that the **m name$_1$** is equivalent to the **m name$_2$**.

Remark: Note that in **SPV**, the string **m name$_1$** should be prior to **m name$_2$** according to the dictionary sort. For example:

equal(na, local(alice, na, Na)); √

equal(local(alice, na, Na), na); X

know is the formula of specification:

- **formula ::= atomic_formula | (formula) | formula + formula | formla * formula | formula > formula | know(agent, (formula))**
- **know(agent name, (formula name)):** Indicate that from the **agent name**'s knowledge, the **agent name** can observe that the **formula name** holds.

The format of **#specification** section is:

#specification

know(agent name$_1$, (formula name$_1$));

know(agent name$_2$, (formula name$_2$));

…

For example:

#specification

know(alice, (said(bob, na)));

know(alice, equal(na, local(alice, na, Na)));

know(bob, know(alice,(said(bob, na))));

Here the property such as

know(alice, (said(bob, na)))

means that **alice** observes that **bob** has said **na**.

The property such as

know(alice, equal(na, local(alice, na, Na)))

means that **alice** observes that the **Na** in the local run of **alice** with nonce **na** is exactly **na**. This is a more intuitive way in authentication.

The property such as

know(bob, know(alice,(said(bob, na))))

means that **bob** observes that **alice** have observed that it is **bob** who said the nonce **na**. This kind of nested specifications is a rather strong assertion since it concerns about other agents' observations from the reasoning of one agent's observations.

5.2 Advanced features

This part is for the advanced users who have more understanding about the Logic of Local Sessions. The **SPV** has given more features and flexibilities in the user-interface for them.

5.2.1 #macro

Macros are supported in **SPV**. Users can define macros by themselves, and this simplifies the input process for big protocols. The Macro definitions begin with **#macro** and are usually put at the beginning of the input file:

#macro

macro definition$_1$;

macro definition$_2$;

For example:

#macro

HOD = cryp(hash, (OrderDesc, PurchAmt));

5.2.2 #define

The **#define** section allows users define their assumptions a little different from the default situation in the Logic of Local Sessions. It begins with **#define**.

SPV permits the users to decide in the following way whether they want to accept the assumption that agents should announce to other agents if there exists an uncompleted local run of the former agents in protocol. So, it is commonly known whether such a local run exists:

normal_state = 0;

means the users do not accept the announcement assumption.

normal_state = 1;

means the users accept the announcement assumption.

Remark: Other values are illegal input, and the **default value** is 1.

Before the specification verification, each agent's observations should be defined according to the Observational Theory. **SPV** permits the users to define each agent's observations by the following means:

observer_variable(agent_term name$_0$) = {

reach(agent name$_0$, label name$_0$, agent_term name$_0$, l name$_0$);

equal(agent name$_{11}$, local(agent name$_0$, label name$_{11}$, agent_term name$_{11}$));

equal(agent name$_{12}$, local(agent name$_0$, label name$_{12}$, agent_term name$_{12}$));

…

equal(nonce name$_{21}$, local(agent name$_0$, label name$_{21}$, nonce_term name$_{21}$));

equal(nonce name$_{22}$, local(agent name$_0$, label name$_{22}$, nonce_term name$_{22}$));

…

equal(guessable_lable_message name$_{31}$, local(agent name$_0$, label name$_{31}$, guessable_lable_message _term name$_{31}$));

equal(guessable_lable_message name$_{32}$, local(agent name$_0$, label name$_{32}$, guessable_lable_message _term name$_{32}$));

…

};

If the users do not define them, the default observations for each agent in its local run are given:

- The agent itself plays the role bound in the initialization.
- Every **agent_term** is equivalent to the actual **agent** bound in the initialization.
- Every **nonce_term** is equivalent to the actual **nonce** bound in the initialization.
- Every **guessable_lable_message_term** is equivalent to the actual **guessable_lable_message** bound in the initialization.

The example of the **#define** section is as follows:

#difine

normal_state = 0;

observer_variable(M) = {

reach(merry, xid, M, 5);

equal(merry, local(merry, chall_clark, M));

equal(chall_clark, local(merry, chall_merry, Chall_C));

};

SPV can let the users design their own specific definitions for each specification to be verified. For example:

#define **//1**

...

#specification **//2**

...

#define **//3**

...

#specification **//4**

...

The verification of specification 2 will be on the basis of definition 1 and 4 of the 3.

5.2.3 #goal

SPV can verify the security properties in Common Authentication Protocol Specification Language (CAPSL, http://www.csl.sri.com/users/millen/capsl/). The specification in CAPSL format begins with **#goal**.

The two kinds of goals in CAPSL are SECRET and PRECEDES:

- **secret M name:** This security property means that the message term generated by a role is a secrecy between the agents.
- **precedes agent_term name$_1$: agent_term name$_2$ | nonce_term name, M name:** This security property means that in **agent_term name$_2$**'s last state, if the agent plays the role of **agent_term name$_2$** holds **agent_term name$_1$**, **nonce_term name** and **M name**, then there is or was an agent plays the role of **agent_term name$_1$** holding the same values of **agent_term name$_2$**, **nonce_term name** and **M name**.

Remark: The **precedes goals** above is equivalent to the following formulas:

reach(agent name$_2$, label name, agent_term name$_2$, l$_{agent_term_name2}$) * equal(agent name$_1$, local(agent name$_2$, label name, agent_term name$_1$)) * equal(nonce name, local(agent name$_2$, label name, nonce_term name))) > (role(agent name$_1$, nonce name, agent_term name$_1$) * equal(agent

name$_2$, local(agent name$_1$, nonce name, agent_term name$_2$)) * equal(nonce name, local(agent name$_1$, nonce name, nonce_term name)) * equal(local(agent name$_1$, nonce name, M name), local(agent name$_2$, label name, M name)))

where the **l$_{agent_term_name2}$** is the length of **agent_term name$_2$**'s local protocol, which is the **agent_term name$_2$**'s local history of sent and received messages described by the protocol body.

- **precedes agent_term name$_1$: agent_term name$_2$ | guessable_lable_term name, M name:** The meaning of this security property is similar to the former property except that the 'nonce' is instead of 'guessable_label_term'.

The format of **#goal** section is:

#goal

secret M name$_{11}$;

secret M name$_{12}$;

…

precedes agent_term name$_{21}$: agent_term name$_{22}$ | nonce_term name$_{21}$, M name$_{21}$;

precedes agent_term name$_{23}$: agent_term name$_{24}$ | nonce_term name$_{22}$, M name$_{22}$;

…

precedes agent_term name$_{31}$: agent_term name$_{32}$ | guessable_lable_term name$_{31}$, M name$_{31}$;

precedes agent_term name$_{33}$: agent_term name$_{34}$ | guessable_lable_term name$_{32}$, M name$_{32}$;

…

For example:

#goal

secret Na;

secret Nb;

precedes A: B | Na, Nb;

6 Reference – Output formats

This section introduces the main parts of a normal output of **SPV**.

6.1 File header

The beginning of an output shows the time required to generate all constrains, and the number of variables and constrains.

6.2 Results of #specification section

The second part is the verification result of the **#specification** section in input file.

For each specification **know(agent name, (formula name))**, the format of output is as follows:

know(agent name, (formula name))

result

=======

We spend seconds to verify this specifications!

where the **result** is **'OK'** or **'NO'**.

6.3 Results of #goal section

The third part is the verification result of the **#goal** section in input file.

6.3.1 Results of SECRET

For each **secret** goal **secret M name**, the format of output is as follows:

==========*******secrecy*************==========**

---------- secret M name ----------

result

We spend seconds to verify this specifications!

6.3.2 Results of PRECEDES

For each **precedes** goal **precedes agent_term name$_1$: agent_term name$_2$ | nonce_term name, M name**, the format of output is as follows:

==========*******precedes*************==========**

--- reach(agent name$_2$, label name, agent_term name$_2$, l$_{agent_term_name2}$) * equal(agent name$_1$,

local(agent name$_2$, label name, agent_term name$_1$)) * equal(nonce name, local(agent name$_2$, label name, nonce_term name))) > (role(agent name$_1$, nonce name, agent_term name$_1$) * equal(agent name$_2$, local(agent name$_1$, nonce name, agent_term name$_2$)) * equal(nonce name, local(agent name$_1$, nonce name, nonce_term name)) * equal(local(agent name$_1$, nonce name, M name), local(agent name$_2$, label name, M name)))---

result

We spend seconds to verify this specifications!

For each **precedes** goal **agent_term name$_2$ | guessable_lable_term name, M name**, the format of output is similar to the former output except that the ‘nonce’ is instead of ‘guessable_label_term’.

7 Reference – Results: experiments

SPV has been tested on several protocols from the Security Protocols Open Repository (http://www.lsv.ens-cachan.fr/spore/index.html). We present only the partial data here; for more details on the tool, please refer to the **…/spv/res** directory. The experiments were conducted on a PC with AMD Opteron242, 2GB DDR memory, running under Redhat Linux 7.0 with gcc 3.2.2.

7.1 Efficiency and scaling

Table 1 summarizes the results of our preliminary experiments in terms of the number of variables, properties generated and time required for some well-known protocols.

Protocol	Variables	Properties	Time (s)
Needham-Schroeder	3548	6296	53.485
Rev Needham-Schroeder	3548	6332	49.609
Den-Sacco shared key	2178	2905	25.045
Woo and Lam Pi	1984	3248	26.214
Kao Chow Authen v1	8197	14316	330.086
Kerberos V5	8834	14946	327.585
The Simplified SET purchase	61065	83747	31364
The SET purchase	147839	222563	224630

Table 1: The number of variables, properties generated and time required in **SPV** verification

7.2 Verified specifications

SPV can verify a lot of complex security properties. Some CAPSL specifications experimental results are shown in Table 2. We use the notation $\mathbf{P^A_B(N,M)}$ to refer to **'PRECEDES A : B | N, M'**. As shown in the following table, the confirmed specifications are signed by **'+'**, and the rest by **'—'**.

Protocol	SECRET	PRECEDES
Needham-Schroeder	$+N_a - N_b$	$-\mathbf{P}_B^A(N_a,N_b) + \mathbf{P}_A^B(N_a,N_b)$
Rev Needham-Schroeder	$+N_a + N_b$	$+\mathbf{P}_B^A(N_a,N_b) + \mathbf{P}_A^B(N_a,N_b)$
Den-Sacco shared key	$+K_{ab}$	$-\mathbf{P}_B^A(T_s,K_{ab}) - \mathbf{P}_A^B(T_s,K_{ab})$
Woo and Lam Pi	$-N_b$	$+\mathbf{P}_B^A(N_b) - \mathbf{P}_A^B(N_b)$
Kao Chow Authen v1	$-N_a - N_b + K_{ab}$	$+\mathbf{P}_B^A(N_a,N_b) + \mathbf{P}_A^B(N_a,N_b)$
Kerberos V5	$+K_{ab}$	$+\mathbf{P}_B^A(T_s,K_{ab}) + \mathbf{P}_A^B(T_s,K_{ab})$
The SET purchase	+PurchAmt	$+\mathbf{P}_P^C$ (XID,CardSecret)

Table 2: Some verified CAPSL specifications of **SPV**

Also, let $\mathbf{K^1_{AB}N}$ stand for some one-level epistemic specification $\mathbf{O_a}$ > **know(a, (said(b, m$_1$) +**

sees(b, m_1))), where **a** plays **A** and runs a local session with **b** plays **B** and m_1 for **N**, and O_a express **a**'s observations when reaching the final step of the protocol. Similarly, $K^2_{AB}N$ stands for two level ones O_a > **know(a, know(b, (said(a,** m_2**) + sees(a,** m_2**))))** , where **a**, **A**, **b**, **B** and O_a are the same as above and m_2 is for **N** in a corresponding local session of **b**. The partial epistemic specifications we have verified are shown in the table 3.

Protocol	EPISTEMIC GOALS
Needham-Schroeder	$+\mathbf{K}^1_{AB}N_a + \mathbf{K}^2_{AB}N_b - \mathbf{K}^2_{BA}N_a$
Rev Needham-Schroeder	$+\mathbf{K}^2_{AB}N_b + \mathbf{K}^2_{BA}N_a$
Den-Sacco shared key	$-\mathbf{K}^1_{AB}T_s - \mathbf{K}^1_{BA}T_s$
Woo and Lam Pi	$-\mathbf{K}^1_{AB}N_b + \mathbf{K}^1_{BA}N_b - \mathbf{K}^2_{AB}N_b$
Kao Chow Authen v1	$+\mathbf{K}^2_{AB}N_b + \mathbf{K}^2_{BA}N_a$
Kerberos V5	$+\mathbf{K}^2_{AB}T_a + \mathbf{K}^2_{BA}T_a$
The SET purchase	$+\mathbf{K}^1_{CM}$ Chall_C$-\mathbf{K}^2_{CM}$ Chall_M + $\mathbf{K}^2_{MC}$ Chall_C

Table 3: Some verified epistemic specifications of **SPV**

As for the SET purchase protocol, we notice that we do not intend to offer more detailed analysis as done in Isabelle; what we want know from these experiments is whether an automatic justification-oriented approach can be scaled to the size of the SET purchase phase protocol.

The experiments results show that **SPV** is powerful and friendly to users. It is open, flexible and scalable. We believe that this tool will be an interesting contribution to the security verification and formal methods communities.